General Unix File API

open	opens file for access (read/write/append/etc)

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)
chown	change uid/gid of file

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)
chown	change uid/gid of file
utime	set time of last modification/access time stamps

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)
chown	change uid/gid of file
utime	set time of last modification/access time stamps
link	create a hard link to a file

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)
chown	change uid/gid of file
utime	set time of last modification/access time stamps
link	create a hard link to a file
unlink	delete a hard link to a file (this may delete the file)

General Unix File API

open	opens file for access (read/write/append/etc)
creat	creates a file (write-only)
close	terminates file access (opposite of open)
read	reads data from file into program's buffer
write	writes data to file
lseek	programmer controls where the next file access occurs
stat,fstat	query file attributes
chmod	change access permissions (read/write/execute)
chown	change uid/gid of file
utime	set time of last modification/access time stamps
link	create a hard link to a file
unlink	delete a hard link to a file (this may delete the file)
umask	sets default file creation mask

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file
O_EXCL|O_CREAT	opens new file, fails if file already exists

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file
O_EXCL|O_CREAT	opens new file, fails if file already exists
O_TRUNC	discards pre-existing file contents

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file
O_EXCL|O_CREAT	opens new file, fails if file already exists
O_TRUNC	discards pre-existing file contents
O_NONBLOCK	subsequent reads/writes should not block

open()

int open(char *path,int access,mode_t permission)

path a string holding a full or relative path

access

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file
O_EXCL|O_CREAT	opens new file, fails if file already exists
O_TRUNC	discards pre-existing file contents
O_NONBLOCK	subsequent reads/writes should not block
O_NOCTTY	don't use named terminal device file as calling process' control terminal

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU read, write, execute (user)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)
S_IWGRP	write (group)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)
S_IWGRP	write (group)
S_IXGRP	execute (group)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)
S_IWGRP	write (group)
S_IXGRP	execute (group)
S_IROTH	read (other)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)
S_IWGRP	write (group)
S_IXGRP	execute (group)
S_IROTH	read (other)
S_IWOTH	write (other)

open()

permission the permission flag is used only with O_CREATs (otherwise, use zero). The following flags may be used in bitwise-or expressions.

S_IRWXU	read, write, execute (user)
S_IRUSR	read (user)
S_IWUSR	write (user)
S_IXUSR	execute (user)
S_IRGRP	read (group)
S_IWGRP	write (group)
S_IXGRP	execute (group)
S_IROTH	read (other)
S_IWOTH	write (other)
S_IXOTH	execute (other)

(see opencreat.c, openexcl.c, and open.c)

umask

mode_t umask(mode_t newmask)

- Specifies permission bits to be masked off.

umask

mode_t umask(mode_t newmask)

- Specifies permission bits to be masked off.
- Subsequent open()'s permission bits are modified according to permission = open's permission & ~mask.

umask

mode_t umask(mode_t newmask)

- Specifies permission bits to be masked off.
- Subsequent open()'s permission bits are modified according to permission = open's permission & ~mask.
- Using umask via a shell *(ksh, bash, etc)* is used to set up default permissions.

umask

mode_t umask(mode_t newmask)

- Specifies permission bits to be masked off.
- Subsequent open()'s permission bits are modified according to permission = open's permission & ~mask.
- Using umask via a shell *(ksh, bash, etc)* is used to set up default permissions.
- Returns the previous value of mask

umask

mode_t umask(mode_t newmask)

- Specifies permission bits to be masked off.
- Subsequent open()'s permission bits are modified according to permission = open's permission & ~mask.
- Using umask via a shell *(ksh, bash, etc)* is used to set up default permissions.
- Returns the previous value of mask

Example 1 **umask(S_RGRP|S_WGRP|S_XGRP)**

The read,write,execute bits associated with the group are masked off (ie. default permissions will be rwx—rwx).

creat()

int creat(const char *path,mode_t mode)

This function is equivalent to

open(path,O_WRONLY|O_CREAT|O_TRUNC,mode)

In earlier versions of Unix, the open() function's access parameter did not handle O_CREAT or O_TRUNC, so open() could only be used on pre-existing files.

Consequently, this function (creat()) really is no longer needed.

read()

ssize_t read(int fd, void *buf, size_t size)

- The fd is generated by open(), creat, or fileno(FILE *).

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.
- Will return qty bytes *actually read*, which may differ from size, especially when used with sockets.

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.
- Will return qty bytes *actually read*, which may differ from size, especially when used with sockets.
- Reads both binary or text.

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.
- Will return qty bytes *actually read*, which may differ from size, especially when used with sockets.
- Reads both binary or text.
- Note that fd need not be a file descriptor for an actual file, but may be a pipe or socket.

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.
- Will return qty bytes *actually read*, which may differ from size, especially when used with sockets.
- Reads both binary or text.
- Note that fd need not be a file descriptor for an actual file, but may be a pipe or socket.
- Read may *block* if no data is currently available from the “file” (ie. pipe, socket).

  (there is a way to prevent blocking behavior)

read()

```
ssize_t read(int fd, void *buf, size_t size)
```

- The fd is generated by open(), creat, or fileno(FILE *).
- This function will (attempt to) read size bytes.
- Will return qty bytes *actually read*, which may differ from size, especially when used with sockets.
- Reads both binary or text.
- Note that fd need not be a file descriptor for an actual file, but may be a pipe or socket.
- Read may *block* if no data is currently available from the “file” (ie. pipe, socket).
  (there is a way to prevent blocking behavior)
- Reads occur at the current file offset, which will be updated by read().

write()

ssize_t write(int fd, const void *buf, size_t size)

- The fd generated by open(), creat(), etc.

write()

```
ssize_t write(int fd, const void *buf, size_t size)
```

- The fd generated by open(), creat(), etc.
- Will attempt to write size bytes, but it may not succeed.

  (sockets: buffers are full, files: hard disk is full, etc)

write()

ssize_t write(int fd, const void *buf, size_t size)

- The fd generated by open(), creat(), etc.
- Will attempt to write size bytes, but it may not succeed.

  (sockets: buffers are full, files: hard disk is full, etc)
- This function may block *(there is a way to set this to non-blocking)*

write()

ssize_t write(int fd, const void *buf, size_t size)

- The fd generated by open(), creat(), etc.
- Will attempt to write size bytes, but it may not succeed.

  (sockets: buffers are full, files: hard disk is full, etc)
- This function may block *(there is a way to set this to non-blocking)*
- Writes occur at the current file offset, which will also be updated after the write.

close()

int close(int fd)

- Returns 0:success -1:failure (and sets errno)

close()

int close(int fd)

- Returns 0:success -1:failure (and sets errno)
- Releases system resources supporting the file descriptor

close()

int close(int fd)

- Returns 0:success -1:failure (and sets errno)
- Releases system resources supporting the file descriptor
- If the process terminates w/o closing, Unix will close those file descriptors itself.

close()

int close(int fd)

- Returns 0:success -1:failure (and sets errno)
- Releases system resources supporting the file descriptor
- If the process terminates w/o closing, Unix will close those file descriptors itself.
- Releases any record locks the process may have on the file

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

  SEEK_CUR relative to current file offset

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

SEEK_CUR	relative to current file offset
SEEK_SET	relative to beginning of file

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

SEEK_CUR	relative to current file offset
SEEK_SET	relative to beginning of file
SEEK_END	relative to end-of-file

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

SEEK_CUR	relative to current file offset
SEEK_SET	relative to beginning of file
SEEK_END	relative to end-of-file

- These three "whence" parameters are defined in unistd.h

(see openseek.c)

lseek()

off_t lseek(int fd, off_t offset, int whence)

- The whence parameters control how lseek does its job

SEEK_CUR	relative to current file offset
SEEK_SET	relative to beginning of file
SEEK_END	relative to end-of-file

- These three "whence" parameters are defined in unistd.h

(see openseek.c)

- Writing past the end-of-file *is permitted!*.

(see cheesey.c)

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd F_GETFL returns current access control flags

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd	F_GETFL	returns current access control flags
	F_SETFL	O_APPEND O_NONBLOCK O_NDELAY

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd	F_GETFL	returns current access control flags
	F_SETFL	O_APPEND O_NONBLOCK O_NDELAY
	F_GETFD	return close-on-exec flag (default is off)

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd	F_GETFL	returns current access control flags
	F_SETFL	O_APPEND O_NONBLOCK O_NDELAY
	F_GETFD	return close-on-exec flag (default is off)
		specifies that a call to exec will close the file

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd	F_GETFL	returns current access control flags
	F_SETFL	O_APPEND O_NONBLOCK O_NDELAY
	F_GETFD	return close-on-exec flag (default is off)
		specifies that a call to exec will close the file
	F_SETFD	0=clear, 1=set close-on-exec flag

fcntl()

```
int fcntl(int fd,int cmd)
int fcntl(int fd,int cmd,long arg)
int fcntl(int fd,int cmd,struct flock *lock)
```

cmd	F_GETFL	returns current access control flags
	F_SETFL	O_APPEND O_NONBLOCK O_NDELAY
	F_GETFD	return close-on-exec flag (default is off)
		specifies that a call to exec will close the file
	F_SETFD	0=clear, 1=set close-on-exec flag
	F_DUPFD	arg indicates file descriptor to be
		duplicated ($\text{fdesc}_{dup} \geq \text{fdesc}_{orig}$).
		Duplicates file pos'n but they may have different
		descriptor flags.
		NOTE: close-on-exec flag *not* duplicated!

fcntl(), con't.

Example 2 *to set or clear O_APPEND file descriptor fd:*

current: `int curflag = fcntl(fd,F_GETFL);` *get current flags*

fcntl(), con't.

Example 2 *to set or clear O_APPEND file descriptor fd:*

current: `int curflag = fcntl(fd,F_GETFL);` *get current flags*

set : `fcntl(fd,curflag|O_APPEND);` *set with* `O_APPEND`

fcntl(), con't.

Example 2 *to set or clear O_APPEND file descriptor fd:*

current:	`int curflag = fcntl(fd,F_GETFL);`	*get current flags*
set :	`fcntl(fd,curflag|O_APPEND);`	*set with* `O_APPEND`
unset :	`fcntl(fd,curflag&(~O_APPEND));`	*unset* `O_APPEND`

fcntl(), con't.

Example 2 *to set or clear O_APPEND file descriptor fd:*

current:	`int curflag = fcntl(fd,F_GETFL);`	*get current flags*
set	`: fcntl(fd,curflag|O_APPEND);`	*set with* `O_APPEND`
unset	`: fcntl(fd,curflag&(~O_APPEND));`	*unset* `O_APPEND`

(see fcntlget.c; illustrates access() and fcntl)

(with F_GETFL, F_GETFD, and F_GETLK)

link(), unlink()

int link(const char *cur_link,const char *new_link)

Creates hard links, so it canot be used across file systems.

cur_link	path to existing file
new_link	path of file's new link

link(), unlink()

int link(const char *cur_link,const char *new_link)

Creates hard links, so it canot be used across file systems.

cur_link	path to existing file
new_link	path of file's new link

int unlink(con st char *cur_link)

Usually this function will delete a file, but for those files with multiple links, then such files will remove the link and decrement the link counter.

Cannot remove directories using unlink without superuser privileges.

rename()

int rename(const char *oldpath, const char *newpath)

- Only works on the same filesystem

rename()

int rename(const char *oldpath, const char *newpath)

- Only works on the same filesystem
- May be used to rename files

rename()

int rename(const char *oldpath, const char *newpath)

- Only works on the same filesystem
- May be used to rename files
- Even to move them from one directory to another

  (remember the same filesystem restriction)

rename()

int rename(const char *oldpath, const char *newpath)

- Only works on the same filesystem
- May be used to rename files
- Even to move them from one directory to another

  (remember the same filesystem restriction)
- If oldpath and newpath are actually hard links to the same file, then rename() does nothing (successfully!)

File Status Functions

Inode, mode, owner, etc – these things can be determined for a file using *status functions*:

File Status Functions

Inode, mode, owner, etc – these things can be determined for a file using *status functions*:

status given path

```
int stat(const char *path, struct stat *buf)
```

File Status Functions

Inode, mode, owner, etc – these things can be determined for a file using *status functions*:

status given path

```
int stat(const char *path, struct stat *buf)
```

status given file descriptor

```
int fstat(int fd, struct stat *buf)
```

File Status Functions

Inode, mode, owner, etc – these things can be determined for a file using *status functions*:

status given path

```
int stat(const char *path, struct stat *buf)
```

status given file descriptor

```
int fstat(int fd, struct stat *buf)
```

status of symbolic link itself

```
int lstat(const char *path, struct stat *buf)
```

File Status Functions

Inode, mode, owner, etc – these things can be determined for a file using *status functions*:

status given path

```
int stat(const char *path, struct stat *buf)
```

status given file descriptor

```
int fstat(int fd, struct stat *buf)
```

status of symbolic link itself

```
int lstat(const char *path, struct stat *buf)
```

(man 2 stat)

(returns 0=success, -1=failure)

File Status Structure

status structure in detail

```
struct stat {
```

File Status Structure

status structure in detail

```
struct stat {
        dev_t st_dev; // ID of device containing file
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize;// blocksize for file system I/O
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize;// blocksize for file system I/O
    blkcnt_t st_blocks; // number of 512B blocks allocated
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize;// blocksize for file system I/O
    blkcnt_t st_blocks; // number of 512B blocks allocated
    time_t st_atime; // time of last access
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize;// blocksize for file system I/O
    blkcnt_t st_blocks; // number of 512B blocks allocated
    time_t st_atime; // time of last access
    time_t st_mtime; // time of last modification
```

File Status Structure

status structure in detail

```
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize;// blocksize for file system I/O
    blkcnt_t st_blocks; // number of 512B blocks allocated
    time_t st_atime; // time of last access
    time_t st_mtime; // time of last modification
    time_t st_ctime; // time of last status change
    };
```

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m) is it a regular file?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	is it a FIFO (named pipe)?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	is it a FIFO (named pipe)?
S_ISLNK(m)	is it a symbolic link?

File Status : Mode

The st_mode contains descriptive information about the file; there are a number of macros which may be used in conjunction with st_mode.

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	is it a FIFO (named pipe)?
S_ISLNK(m)	is it a symbolic link?
S_ISSOCK(m)	is it a socket?

File Status: RWX

The st_mode also contains permission information:

File Status: RWX

The st_mode also contains permission information:

S_IRWXU mask for file owner permissions

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission
S_ISUID	setuid bit *(see next slide)*

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission
S_ISUID	setuid bit *(see next slide)*
S_ISGID	setgid bit *(see next slide)*

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission
S_ISUID	setuid bit *(see next slide)*
S_ISGID	setgid bit *(see next slide)*
S_ISVTX	sticky bit *(see next slide)*

File Status: RWX

The st_mode also contains permission information:

S_IRWXU	mask for file owner permissions
S_IRUSR	owner has read permission
S_IWUSR	owner has write permission
S_IXUSR	owner has execute permission
S_IRWXG	mask for group permissions
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	mask for permissions for others (not in group)
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission
S_ISUID	setuid bit *(see next slide)*
S_ISGID	setgid bit *(see next slide)*
S_ISVTX	sticky bit *(see next slide)*

For example, to determine if the owner has read and write permission:

```
statbuf.st_mode & (S_IRUSR|S_IWUSR)
```

Special File Permissions

- Normally when executing a file, the user's permissions (read-write-execute) are inherited by the running program.
  Ie. when one runs a program owned by root, the program has the user's permissions, not root's permissions, in effect.

Special File Permissions

- Normally when executing a file, the user's permissions (read-write-execute) are inherited by the running program.
  Ie. when one runs a program owned by root, the program has the user's permissions, not root's permissions, in effect.
- However, if the file has the **setuid** permission bit set, then the file's owner's permissions are inherited by that program. The running program's *effective user-id* becomes that of the file's owner, not the user.
  To set the setuid bit from the command line: *chmod u+s filename(s)*
  Use `chmod(path,S_ISUID|...)` to set the setuid bit in a program.

Special File Permissions

- Normally when executing a file, the user's permissions (read-write-execute) are inherited by the running program.
  Ie. when one runs a program owned by root, the program has the user's permissions, not root's permissions, in effect.
- However, if the file has the **setuid** permission bit set, then the file's owner's permissions are inherited by that program. The running program's *effective user-id* becomes that of the file's owner, not the user.
  To set the setuid bit from the command line: *chmod u+s filename(s)*
  Use `chmod(path,S_ISUID|...)` to set the setuid bit in a program.
- One may also have the **setgid** bit set, which means that the *effective group-id* becomes that of the file's group, rather than the user's group.
  To set the setuid bit from the command line: *chmod g+s filename(s)*
  Use `chmod(path,S_ISGID|...)` to set the setgid bit in a program.

Special File Permissions, con't.

- The **sticky bit** prevents unprivileged users from removing or renaming a file in a directory, even though it is world-writable. The sticky bit is found, for example, on the */tmp* directory.

  Use *chmod +t directoryname* from the command line on a directory name to set this bit.

  Use `chmod(path,S_ISVTX|S_ROTH|S_IWOTH|S_IXOTH)` to make a directory have rwx and sticky bit for others in a program.

Special File Permissions, con't.

- The **sticky bit** prevents unprivileged users from removing or renaming a file in a directory, even though it is world-writable. The sticky bit is found, for example, on the */tmp* directory.

  Use *chmod +t directoryname* from the command line on a directory name to set this bit.

  Use `chmod(path,S_ISVTX|S_ROTH|S_IWOTH|S_IXOTH)` to make a directory have rwx and sticky bit for others in a program.

Numerically, the setuid, setgid, and sticky bits are represented as:

Value	*Explanation*
0	setuid, setgid, sticky bits are unset
1	sticky bit is enabled
2	setgid bit is enabled
3	setgid and sticky bits are enabled
4	setuid bit is enabled
5	setuid and sticky bits are enabled
6	setuid and setgid bits are enabled
7	setuid, setgid, sticky bits are enabled

Special File Permissions, con't.

- The **sticky bit** prevents unprivileged users from removing or renaming a file in a directory, even though it is world-writable. The sticky bit is found, for example, on the */tmp* directory.

  Use *chmod +t directoryname* from the command line on a directory name to set this bit.

  Use `chmod(path,S_ISVTX|S_ROTH|S_IWOTH|S_IXOTH)` to make a directory have rwx and sticky bit for others in a program.

Numerically, the setuid, setgid, and sticky bits are represented as:

Value	*Explanation*
0	setuid, setgid, sticky bits are unset
1	sticky bit is enabled
2	setgid bit is enabled
3	setgid and sticky bits are enabled
4	setuid bit is enabled
5	setuid and sticky bits are enabled
6	setuid and setgid bits are enabled
7	setuid, setgid, sticky bits are enabled

(having setgid set and setuid not set can have special meaning; see next class)

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.
- The mode can be a bitwise OR'd combination of the following:

F_OK	does the file exist?

access()

```
int access(const char *pathname, int mode)
```

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.
- The mode can be a bitwise OR'd combination of the following:

F_OK	does the file exist?
R_OK	does the file exist and grant read permission?

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.
- The mode can be a bitwise OR'd combination of the following:

F_OK	does the file exist?
R_OK	does the file exist and grant read permission?
W_OK	does the file exist and grant write permission?

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.
- The mode can be a bitwise OR'd combination of the following:

F_OK	does the file exist?
R_OK	does the file exist and grant read permission?
W_OK	does the file exist and grant write permission?
X_OK	does the file exist and grant execute permission?

access()

int access(const char *pathname, int mode)

- With access(), the calling function can determine if it can access the file in pathname.
- Symbolic pathnames are dereferenced
- The check is done with the process' real UID and GID, rather than the effective UID, GID.
- On success, 0 is returned; on error, -1 is returned, and errno is set.
- The mode can be a bitwise OR'd combination of the following:

F_OK	does the file exist?
R_OK	does the file exist and grant read permission?
W_OK	does the file exist and grant write permission?
X_OK	does the file exist and grant execute permission?

Thus one may check for read and write permission simultaneously by using R_OK|W_OK

chmod(), fchmod()

```
int chmod(const char *path, mode_t mode)
int fchmod(int fd, mode_t mode)
```

- Use these functions to change permissions on a file

chmod(), fchmod()

```
int chmod(const char *path, mode_t mode)
int fchmod(int fd, mode_t mode)
```

- Use these functions to change permissions on a file
- chmod() works via a pathname (string)

chmod(), fchmod()

```
int chmod(const char *path, mode_t mode)
int fchmod(int fd, mode_t mode)
```

- Use these functions to change permissions on a file
- chmod() works via a pathname (string)
- fchmod() works via a file descriptor

chmod(), fchmod()

```
int chmod(const char *path, mode_t mode)
int fchmod(int fd, mode_t mode)
```

- Use these functions to change permissions on a file
- chmod() works via a pathname (string)
- fchmod() works via a file descriptor
- These functions use the same mode macros that open() uses

chmod(), fchmod()

```
int chmod(const char *path, mode_t mode)
int fchmod(int fd, mode_t mode)
```

- Use these functions to change permissions on a file
- chmod() works via a pathname (string)
- fchmod() works via a file descriptor
- These functions use the same mode macros that open() uses

O_RDONLY	read-only access
O_WRONLY	write-only access
O_RDWR	read and write access
O_APPEND	open file for appending
O_CREAT	creates/opens file
O_EXCL|O_CREAT	opens new file, fails if file already exists
O_TRUNC	discards pre-existing file contents
O_NONBLOCK	subsequent reads/writes should not block
O_NOCTTY	don't use named terminal device file as calling process' control terminal

utime()

int utime(const char *filename, const struct utimbuf *times)

- Modifies access and modification times of a file.

utime()

int utime(const char *filename, const struct utimbuf *times)

- Modifies access and modification times of a file.
- The utimbuf structure holds two times...

```
struct utimbuf {
        time_t actime; // access time
        time_t modtime; // modification time
        }
```

utime()

int utime(const char *filename, const struct utimbuf *times)

- Modifies access and modification times of a file.
- The utimbuf structure holds two times...

```
struct utimbuf {
    time_t actime; // access time
    time_t modtime; // modification time
    }
```

- When the times is NULL, then actime and modtime of the file named in filename are set to the current time.

utime()

int utime(const char *filename, const struct utimbuf *times)

- Modifies access and modification times of a file.
- The utimbuf structure holds two times...

```
struct utimbuf {
    time_t actime; // access time
    time_t modtime; // modification time
    }
```

- When the times is NULL, then actime and modtime of the file named in filename are set to the current time.
- Time is measured in seconds relative to the “Epoch”:

  00:00:00 Jan 1, 1970 UTC

  is time “zero”

utime()

int utime(const char *filename, const struct utimbuf *times)

- Modifies access and modification times of a file.
- The utimbuf structure holds two times...

```
struct utimbuf {
    time_t actime; // access time
    time_t modtime; // modification time
    }
```

- When the times is NULL, then actime and modtime of the file named in filename are set to the current time.
- Time is measured in seconds relative to the “Epoch”:

  00:00:00 Jan 1, 1970 UTC

  is time “zero”
- As usual, returns 0=success and -1=failure

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

struct tm *localtime(const time_t *timep)

converts time_t to local tm

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

struct tm *localtime(const time_t *timep)

converts time_t to local tm

struct tm *gmtime(const time_t *timep)

converts time_t to UTC tm

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

struct tm *localtime(const time_t *timep)

converts time_t to local tm

struct tm *gmtime(const time_t *timep)

converts time_t to UTC tm

char *asctime(const struct tm *tm)

like ctime(), but uses struct tm *

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

struct tm *localtime(const time_t *timep)

converts time_t to local tm

struct tm *gmtime(const time_t *timep)

converts time_t to UTC tm

char *asctime(const struct tm *tm)

like ctime(), but uses struct tm *

time_t mktime(struct tm *tm)

converts tm to time_t

Time Functions

#include <time.h>

time_t time(time_t *t)

returns qty seconds since Epoch

char *ctime(const time_t *timep)

returns qty seconds since Epoch, but returns a human-readable string

struct tm *localtime(const time_t *timep)

converts time_t to local tm

struct tm *gmtime(const time_t *timep)

converts time_t to UTC tm

char *asctime(const struct tm *tm)

like ctime(), but uses struct tm *

time_t mktime(struct tm *tm)

converts tm to time_t

clock_t clock(void)

may vary, but ANSI has it as the qty of μsec elapsed since calling process began executing

The Structure of Time

```
struct tm {
        int tm_sec;         /* seconds */
```

qty seconds after the minute, 0-59

The Structure of Time

```
struct tm {
        int tm_sec;     /* seconds */
        int tm_min;     /* minutes */
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */
    int tm_min;     /* minutes */
    int tm_hour;    /* hours */
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */       qty seconds after the minute, 0-59
    int tm_min;     /* minutes */       qty minutes after the hour, 0-59
    int tm_hour;    /* hours */         qty hours past midnight, 0-23
    int tm_mday;    /* day of the month */  in the range of 1-31
```

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */
    int tm_min;     /* minutes */
    int tm_hour;    /* hours */
    int tm_mday;    /* day of the month */
    int tm_mon;     /* month */
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

in the range of 1-31

qty months since January, 0-11

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */
    int tm_min;     /* minutes */
    int tm_hour;    /* hours */
    int tm_mday;    /* day of the month */
    int tm_mon;     /* month */
    int tm_year;    /* year */
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

in the range of 1-31

qty months since January, 0-11

qty years since 1900

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */
    int tm_min;     /* minutes */
    int tm_hour;    /* hours */
    int tm_mday;    /* day of the month */
    int tm_mon;     /* month */
    int tm_year;    /* year */
    int tm_wday;    /* day of the week */
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

in the range of 1-31

qty months since January, 0-11

qty years since 1900

qty days since Sunday, 0-6

The Structure of Time

```
struct tm {
    int tm_sec;     /* seconds */          qty seconds after the minute, 0-59
    int tm_min;     /* minutes */          qty minutes after the hour, 0-59
    int tm_hour;    /* hours */            qty hours past midnight, 0-23
    int tm_mday;    /* day of the month */ in the range of 1-31
    int tm_mon;     /* month */            qty months since January, 0-11
    int tm_year;    /* year */             qty years since 1900
    int tm_wday;    /* day of the week */  qty days since Sunday, 0-6
    int tm_yday;    /* day in the year */  qty days since Jan 1, 0-365
```

The Structure of Time

```
struct tm {
        int tm_sec;     /* seconds */
        int tm_min;     /* minutes */
        int tm_hour;    /* hours */
        int tm_mday;    /* day of the month */
        int tm_mon;     /* month */
        int tm_year;    /* year */
        int tm_wday;    /* day of the week */
        int tm_yday;    /* day in the year */
        int tm_isdst;   /* daylight saving time */
        };
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

in the range of 1-31

qty months since January, 0-11

qty years since 1900

qty days since Sunday, 0-6

qty days since Jan 1, 0-365

positive if DST in effect,

0 if not, negative if unknown

The Structure of Time

```
struct tm {
        int tm_sec;     /* seconds */
        int tm_min;     /* minutes */
        int tm_hour;    /* hours */
        int tm_mday;    /* day of the month */
        int tm_mon;     /* month */
        int tm_year;    /* year */
        int tm_wday;    /* day of the week */
        int tm_yday;    /* day in the year */
        int tm_isdst;   /* daylight saving time */
        };
```

qty seconds after the minute, 0-59

qty minutes after the hour, 0-59

qty hours past midnight, 0-23

in the range of 1-31

qty months since January, 0-11

qty years since 1900

qty days since Sunday, 0-6

qty days since Jan 1, 0-365

positive if DST in effect,

0 if not, negative if unknown

(used by localtime() gmtime() asctime() mktime())