Minix block device drivers all have a similar structure.

In any working Minix, at least two block drivers are compiled:
- memory device driver
- some storage device driver (either floppy or hard disk, usually both)

Earlier versions of Minix had a CDROM driver, but that is nowadays handled by the at_wini (IDE hard disk) driver, since most CDROM are IDE devices (many motherboards allow modern SATA controllers to be seen as IDE controllers if running an older OS)

Each driver has its specific hardware init before its main loop. For the RAM disk driver this involves reserving/acquiring control of memory. For the hard disk this involves determining drive geometry and parameters.

The driver main loop waits for a message, calls the appropriate function to perform the operation requested by the message, then generates a reply message.
Function pointers in C

In C (and C++) it is possible to create a pointer that contains the address of a function (instead of a variable). Functions cannot generally be modified this way, but they can be called by invoking the usual parentheses () call syntax.

The parameter list and return type of the function counts as part of the pointer type. Therefore, a pointer to a function that returns void and takes an int argument is of a different pointer type than one that points to a function that returns int and takes two doubles.

Why is this useful? It allows one to "plug" any function of a certain type into some piece of generic code.

This is typically used for "call-backs".

The idea of treating functions as exchangeable entities leads is the basis of many useful generic-programming techniques.
Block device drivers all use the same 'main loop' function. How can this function call each driver's specific functions? By using function pointers.

Every driver sets up a `driver` struct (defined in `drivers/libdriver/driver.h`) that contains a function pointer for each action the driver can perform (open, close, read, etc).

The generic/shared driver main loop needs such a struct as argument:

```c
message mess;    /* message buffer*/

void shared_io_driver(struct driver table *entry_points){
    /* initialization is done by each driver before calling this */
    while (TRUE) {
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type) {
            case READ: rcode = (*entry_points->dev_read>(&mess); break;
            case WRITE: rcode = (*entry_points->dev_write>(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
            default: rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        mess.status = rcode;    /* result code*/
        send(caller, &mess);
    }
} 
```
The driver itself then sets up this "pointer table" and calls the main loop.

The memory driver shown here is an example.

All of the values that are put into the m_dtab struct are names (hence addresses) of functions defined in the rest of the memory.c file.
Generic driver main loop

Receive a pointer to a struct, which contains pointers to functions

```c
message mess;

void shared_io_task(struct driver_table *entry_points) {
    /* message buffer */
    while (TRUE) {
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type) {
            case READ: rcode = (*entry_points->dev_read(&mess)); break;
            case WRITE: rcode = (*entry_points->dev_write(&mess)); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
            default: rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode; /* result code */
        send(caller, &mess);
    }
}
```

Message structure (contains sender, process ID, request type, etc.)
Invoke the function pointer that points to the correct function for reading the calling driver’s physical device
Block operations overview

- Operations that can be requested from a device driver
  - Correspond to the possible values of the message’s m.m_type field

- Six operations can be requested
  1. OPEN
  2. CLOSE
  3. READ
  4. WRITE
  5. IOCTL
  6. SCATTERED_IO
Block driver operations

- The meanings of **READ** and **WRITE** are fairly clear
  - A block of data is transferred
    - From the device
    - To the memory of the process that initiated the call
    - Or vice versa
- The **OPEN** and **CLOSE** operations have similar meanings to
  - The way **open** and **close** system calls apply to operations on files
    - An **OPEN** operation should
      - Verify that the device is accessible, or
      - Return an error message if not
    - A **CLOSE** operation should guarantee that
      - Buffered data written by the caller is transferred to the device
Block driver operations

- The **IOCTL** operation ("IO control")
  - Many I/O devices have operational parameters
  - These must occasionally be examined and perhaps changed
    - The **IOCTL** operations handle this
    - For example, examining or changing how a disk is partitioned
    - Entering low-power state when suspending the computer
    - Opening or closing the CDROM drive

- The **SCATTERED_IO** operation
  - Allows the **file system** to request a multi-block read or write with a single command.
  - In the case of a **READ** operation
    - The additional blocks may be read, even if not requested by the user process (the file system can attempt to anticipate future requests)
  - Several **WRITE** operations may be buffered
    - Requests kept in a buffer are not immediately written to disk.
    - Helps speed up writes.
Minix is not, in general as platform-independent as more developed operating systems such as Linux or FreeBSD, ie. Some Minix code is written for specific hardware, notably IBM/Intel-compatible PC hardware (CPU, DMA controller, IDE).

The files `drvlib.c` and `drvlib.h` contain code written specifically for IBM PC based computers, and will not work on other types of machines. Fortunately, most desktop and laptop PCs are based on the IBM architecture, so this is not usually a problem in practice. It would be possible to write drivers for other systems as well.
IO in Minix

Every single R/W request is eventually handled by do_rdwt (driver.c:149) which is a generic function used by all block drivers.

This function first does basic checks such as if transfer buffers are OK and the partition being accessed is valid.

It calls the specific driver's dr_prepare function.

Then calls the driver's dr_transfer function which will manipulate the actual controller registers to cause certain blocks to be read from the device.

The driver's must generally be aware of disk geometry ie. how many platters, tracks, cylinders, sectors etc. the disk is divided into and in what chunks the controller can read/write data.

Of course, this is not relevant for something like the RAM disk driver.
Hard drives (and some other storage media) are usually partitioned.

Noun: partition (plural partitions)
1. An action which divides a thing into parts, or separates one thing from another.
2. A part of something that has been divided.

... 
3. (computing) A section of a hard disk separately formatted.

Each partition is separately formatted with a file system structure.

Although multiple partitions reside on one disk, each is treated (from user code's perspective) as a different device.

Partitioning is useful for keeping OSs separate, or keeping parts of an OS separate (eg. /home and /).
There are several reasons for partitioning disks into smaller pieces. Unfortunately, one reason is that some OS (such as book version of Minix) can only handle drives up to a certain size.

Minix can only handle max 4GB file systems. It would be wasteful to use bigger partitions (although not harmful).

(technically this is true for all operating systems, but the limits are generally much higher than what the average user would require at the time)

Another good reason is that different OS or parts of an OS may use different file system formats

Windows uses NTFS, Linux ext3/ext4, and Minix uses Minix FS

Linux has other alternatives, eg. can use ext3 for /etc and reiserFS for /var/log on a webserver

Information about the partitions present on a HD are written to the partition table located at the start of the disk.
Code for recognizing and creating HD partitions is contained in `drvlib.c` instead of `driver.c`.

Not all devices use partitions or all possible IO features; only need the latter.

When a block device is opened for the first time the partition function is called (in `drvlib.c`) This leads to calling `get_part_table` which reads the partition table.

A minor device number is allocated for every partition found (this allows 'binding' each partition to a device file in `/dev` with `mknod` later).

Partition types are identified by a "magic number" in the part table.

Whenever partition detects a Minix filesystem partition, it performs some further work to gather information about that partition.

Partition table information is then copied into a structure in the disk device driver, for further use.
Minix handles multiple partitions with a simple scheme that relies on fixed-sized arrays in the partition table. Windows uses a more flexible linked-list based scheme called "extended partitions". Minix provides limited support for such partitions (but will likely not run on one).

From the book:

"Extpartition (line 11501) has nothing to do with MINIX 3 itself, so we will not discuss details. Some other operating systems (e.g., Windows) use extended partitions. These use linked lists rather than fixed-size arrays to support subpartitions. For simplicity MINIX 3 uses the same mechanism for subpartitions as for primary partitions. However, minimal support for extended partitions is provided to support MINIX 3 commands to read and write files and directories of other operating systems. These operations are easy; providing full support for mounting and otherwise using extended partitions in the same way as primary partitions would be much more complicated."