DOSContainer official documentation

Bas v.d. Wiel

May 3, 2024

Contents

Chapter 1

Introduction

DOSContainer was born out of a dissatisfaction with the way game packs were created for computer platforms on the MiSTerFPGA platform. Many such collections focus on turning computers into consoles, which I felt took away from the intricacies of these old systems. Having quite a good memory of my own DOS days, I decided to codify all of that knowledge into a tool that will run on modern platforms.

I started out with a proof of concept using the Bash shell on a Linux system, because that's also the operating system that runs on the MiSTer. After some interesting weeks I actually finished the project and it still lives on GitLab at the time of this writing. You can use it go generate disk images with it, and they will even work! Maintenance on the script itself quickly turned into a massive nightmare, as did the long list of external dependencies.

This manual covers the Rust version of DOSContainer, which was conceived after I had proven to myself that the idea behind DOSContainer was feasible. Coding the tool in Rust would fix a few major annoyances for me, least of which was the long list of dependencies. By default, Rust spits out statically compiled binaries. This means that you get a single executable file, tailored natively to the platform it was built for, that will just work without any external libraries or dependencies needed. So no DLL hell, not even on Windows!

Another aspect that greatly improved was control. Using Rust I get to control every single bit that gets written to the disk image, as well as a whole set of ready-made libraries that handle things like file downloading, checksums, parsing YAML etc. In short, Rust gives me quality control. It also provides me with a built-in mechanism for testing my code, so I can prove that things work as intended as the codebase sprawls.

The manual you are reading now comes with pre-alpha releases of the tool. That means that the core functionality is not yet completed. Releases start at version 0.0.3 because the first two minors were taken by the Bash version. I'm not developing those anymore, so the 0.0.3 version will be what gets released as the first real alpha.

The feature set on the alpha release will be:

- Create valid VHD-format disk image files. Not even Windows does that!
- Partition and format bootable disks identical to IBM PC-DOS 2.00.
- Install all of IBM PC-DOS 2.00 on the image optionally.
- Install Alley Cat from YAML.

At the time of this writing DOSContainer ticks all the boxes except the third. This hinges on support for subdirectories, which is very badly documented across the web and had to figured out through reverse engineering the result of how DOS itself does it.

Figure 1.1: Introduction screen of Alley Cat.

1.1 About this manual

This documentation is split into parts that cover the operation of DOSContainer in the form of a user manual. This is particularly relevant for those who want to package new games using this toolbox. Running the build process is as simple as learning a single command so that's hardly worth a full manual. The other parts cover the implementation and design of DOSContainer itself. This is part of the ambition to release a *museum quality* product. Sure the code is the documentation in the open source world, but that only documents the *what* and the *how*, not the *why*. The latter is crucial to proper documentation so that's the purpose of this documentation. The last part of this document covers the technical details of the systems DOSContainer seeks to implement. Not because no documentation on ancient file systems exists, but it is frightfully scattered across the web. A primary resource of this information is a set of personal pages created by a retired professor at the Eindhoven University of Technology. While

I tip my hat to Andries Brouwer for his efforts, I have no idea how long the university will keep his pages available. This document aims to closely bundle the contents of his and other pages into a comprehensive reference for the FAT file systems as they were of old, rather than how they are now (because there are differences in practice).

Part I

User manual

Chapter 2

DOSContainer command reference

DOSContainer in its current state is hardly functional at all. Regardless of that, this chapter outlines the command line interface for this program. Depending on the operating system on your computer, you need to use a different binary file. On Windows this file is caled `doscontainer.exe` while on any other platform it's just `doscontainer`. Whenever you see a reference to `doscontainer` in a command line, you should substitute the .exe version if you are on Windows. Functionally all versions are 100% identical.

On Linux and other UNIX-like systems such as macOS an example of the command would look like this:

```
$doscontainer build alleycat.yaml
```

..and on Windows this same example would be something like:

```
C:\Progra~1\DOSK8S\doscontainer.exe build alleycat.yaml
```

2.1 Running commands

DOSContainer's commands are invoked by adding a single word to the invocation of the binary itself, as illustrated above: the word `build` is the command in these examples. DOSContainer in its current pre-alpha form knows two commands:

analyze
Print debugging information on any VHD-format disk image.

build
Constructs a VHD image from an input file in YAML format.

The `analyze` command needs the filename of a VHD-format disk image as a parameter, like so:

```
$doscontainer analyze alleycat.vhd
```

Given a valid VHD-file, it will output all sorts of debugging information on the VHD container format itself, the partition table, the filesystem and the files that are found on the disk image. The reason why this command exists at all, is that it was useful during development. It was left in the tool because tearing it out would take effort, and it could in fact be useful to others as well.

The `build` command is the bread and butter of the DOSContainer tool. The command itself is very simple because it just takes a single YAML file. It generates a VHD-file in the same directory from where you invoked the DOSContainer program. So assuming DOSContainer is in your path somewhere like installed in `/usr/local/bin` on Linux, you could invoke it from anywhere. Let's say you have a `games` folder in your home directory from which you run DOSContainer:

```
$doscontainer build alleycat.yaml
```

or for Windows:

```
C:\Users\JohnDoe\doscontainer.exe build alleycat.yaml
```

..would yield a file named `alleycat.vhd` right in your own home directory. If you happen to be working right on the MiSTer itself, you could run this from `/media/fat/games/PCXT` and you'd be able to mount the new file and use it directly.

Chapter 3

YAML manifest syntax reference

!!THIS PART DOES NOT WORK AS WRITTEN!!

DOSContainer uses YAML as the file format for what it calls manifest files. These files constitute the shopping list of all the parts DOSContainer needs to combine into a working VHD disk image. The format is very common in technical circles to specify all kinds of configuration information. Because it's such a widely used industry standard, DOSContainer adopts it as its default format. The main advantage being that libraries to handle the format are readily available.

3.1 Example of a manifest

The Alley Cat game is a classic released by IBM in 1984. It would be really nice to package this with a period appropriate operating system like IBM PC-DOS 2.00. A manifest for this game could look like this:

```
---
version: 2
metadata:
  title: Alley Cat
  publisher: IBM
  year: 1984
  comment: Testing DOSContainer, not functional yet.
disk:
  name: alleycat.vhd
  type: pcxt
  size: 10
os: IBMDOS200
application:
  - url: "https://dosk8s-dist.area536.com/alleycat.zip"
    checksum: 0316fb862c67fdf9318a5c8513d99e5af185ce10306d20c27f5c6da099b5b176
    label: Alley Cat
```

You'll notice that the file starts with three dashes. That's standard and denotes that what follows is in fact YAML. Beyond that, you should notice that there are differences in indentation. Some lines start a little further to the right than others. This is **important** in YAML so take note. The words that start at the very left of the manifest are what I'll call "sections". Inside these you'll find individual settings on a line that's moved two (no more and no less) spaces to the right. Oh and then there's bits that are indented and start with a dash. Those are written this way because you could have more than one of them, like items on a shopping list.

The *version* field is optional. DOSContainer started life as a Bash script that uses YAML format that is different from the one described here. Hindsight being quite clear indeed, I decided to add a version field to the specification. DOSContainer will treat absence of this field to mean the same as a value of `1`, and will try to parse the input as if it were dealing with the old Bash version. Future breaking changes to the YAML format will be a lot easier to handle with a number to track.

3.2 Metadata

The metadata section contains information that is not used for building the actual VHD file itself. It is there to provide a machine readable form for information about the manifest. This will be useful when it becomes possible to use DOSContainer from a website or through a GUI. You should make a habit of filling the fields for *title*, *publisher*, *year* and optionally *comment* if there's anything non-obvious about this manifest that users should know but can't easily see from reading the rest of it.

3.3 Disk

The disk section bundles settings common to the VHD file that should be generated. The *name* field defines the filename of the VHD on your host system. No need to be compatible with old MS-DOS style naming conventions here, but do make sure you're not doing weird things that your host machine doesn't like. Avoid slashes, backslashes and the like. The tool was not built with support for directories in mind, so any breakage here is your own responsibility.

The *type* field defines the disk type. This hides a little bit of complexity concerning disk geometry and the way hard drives used to work in physical machines. Currently only `pcxt` is supported but others will be added. Leave it at `pcxt` for now and you'll be able to generate VHD's for just about anything as long as it doesn't need more than a little below 32MB of disk space.

This brings us to the *size* field. This is number is the size of the disk in MegaBytes. I realize that this is an incredibly coarse metric for the early 1980's when floppy disks were the common distribution medium for games, and they would rarely be installed onto a hard drive. Those floppies would generally have no more than 360KB's or space, so to give a game 10MB of hard drive may seem like massive overkill and it actually is[1]. On the other hand, we live in the 21st century today and between huge hard drives and clever on-the-fly compression in file systems.. what's 10MB among friends right?

3.4 Operating system

The *os* variable is currently just a single value because there is, as of this writing, nothing to configure yet. You can set the operating system to install here. Currently only the English version of IBM PC-DOS 2.00 is supported, which is signified by the shorthand value of `IBMDOS200`. Many others will follow. Eventually the YAML structure will expand to accommodate tweaks and settings to the operating system like different languages if available. For now, this is it.

3.5 Application

Here we define a list of ZIP-files that get downloaded, extracted and shipped into the final VHD. The *url* field takes the address of your file over either plaintext HTTP or secure HTTPS. You can put your files anywhere you want, as long as DOSContainer is able to reach them. In the example you see a reference to my own environment. If you want something else from DOSContainer, you just set up your own. DOSContainer does not care what kind of server you use. Could be anything: GitHub, your NAS at home, your blog host.. knock yourself out.

You'll notice there's a dash in front of the *url* field and the *checksum* field is aligned flush with the *url* field. That turns this section into a list. You could put multiple zip files into the *application* section and they would all get downloaded and injected into your VHD. This allows you to do things like install add-ons into games, compose packs, even eventually install Windows into DOS if you were so inclined and then layer a game or application right on top of that. Madness!

The *checksum* field is optional. It contains the SHA256 checksum of the zip file as you uploaded it. This is something of a measure to prevent two things. One of these is malicious tampering. The other, much more likely scenario, is some game packager doing something unexpected to their zip file (updating it) causing unexpected things to happen for people using the manifest. Since there is no good way to keep manifests and zipfiles tightly linked together, this checksum is here to make certain that your manifest matches the zip files it refers to.

The *label* field is an optional string that DOSContainer can use, if at all present, when it wants to tell the user something about this particular file. If it's not present, then the URL field will be used but this may well be ugly. So in case of the example you could add `name: Alley Cat` to the manifest. That'd allow DOSContainer to tell the user it's downloading `Alley Cat` instead of some ugly URL nobody wants to see anyway.

[1]DOSContainer may get an extension to support generating real floppy images for these small games, given enough community interest. It's not that hard to do, but not the primary focus.

Part II

Design

Chapter 4

Design and principles

DOSContainer follows a number of design and philosophical principles.

- DOSContainer is **predicated on layers**, much like Docker containers. It pulls in many parts, layers them on top of each other and composes a disk image as the finished result. You can then use that image file to run software on either an emulator or even a real vintage computer, given a way to transfer the image there. The layers ensure that mixing and matching of all components of a software environment is possible. It gives the end user the ultimate flexibility in using the manifest files to output any out of potentially thousands of different permutations without ever touching the DOS environment itself.

- **Native binary, fully self-contained**. This means DOSContainer itself will never be more than a single executable file for any platform. This eases both distribution and use. Just drop the file anywhere on your system and it will just work.

- **Museum quality** is the goal. This means that there should be as little difference between the output of DOSContainer and an actual, vintage computer doing the same thing in real hardware. Absolute accuracy is not always attainable, nor is it always desirable (not every bug is worth having). As an overarching principle, it stands: DOSContainer follows standards like HTTPS, the VHD disk format and YAML in the parts where it interacts with the modern world, and aims for perfect imitation where it mimics vintage hardware.

- DOSContainer is and will always be **Free Software** (capitalized), published under the 3-Clause BSD license.

- DOSContainer is **not intended to facilitate software piracy or other illegal activities**. As an end user you are responsible for your own behavior. DOSContainer embeds very small snippets of x86 Assembler code that were lifted from installed instances of proprietary operating systems.

These snippets are the boot sector loaders that are used to make a disk image bootable when combined with the rest of the operating system's files. These other files are *not included* in DOSContainer. Many websites across the internet offer these files, sometimes in a very high-profile and visible manner. DOSContainer can not work without these system files, but it is up to you to determine whether you are entitled to use these files. Usually you are if you have a valid license. I am, however, not a laywer and you should not take these words as any kind of legal advice. Get proper counsel if you're worried, or simply don't use this software at all. As the author of DOSContainer I am working under the assumption that my use of these tiny proprietary bits is defensible under fair use.

- DOSContainer is **not a commercial enterprise**. Author creates this software as a hobby and this is the only intended use for it. Please don't be an idiot and use DOSContainer to build disk images for the big industrial installations inf your factory. This tool is intended to make it easy for you to play your old games, not to do serious work. The license won't stop you, but don't say I didn't warn you!

Chapter 5

Operating Systems

5.1 IBM DOS 2.00

IBM PC-DOS 2.00 was the first version of DOS that officially supported hard disk drives and was to be paired with the IBM 5160, known colloquially as the IBM XT.

DOSContainer uses this version of DOS as the reference for version 2.00. The original does seem to contain a bug that's hard to replicate as well as unwanted. When formatting a disk, the `FORMAT` command seems to inject a number of seemingly random bytes towards the end of the FAT tables. This could lead to the OS erroneously thinking we're running out of space while that isn't the case at all. I chose not to implement this behavior as it's absolutely detrimental to the user experience.

Part III

Background documentation

Chapter 6

Disk drives and virtual images

DOSContainer concerns itself with disk image files. These are modern, virtual recreations of what used to be physical hard drives. For that reason I'm going to expand a bit on what a physical hard drive was like in the 1980's. It helps to understand what you're dealing with before trying to use it, right?

Physical hard drives all have a geometry that can be expressed as a triplet consisting of Cylinders, Heads and Sectors-per-track. For a mental picture of how this works, imagine a hard drive to be like a vinyl record player. Instead of a single needle that traces the groove on a single side of a record, we deal with a stack of magnetic disks that each have the equivalent of a needle on both sides. A typical hard disk in the IBM XT era would have two rotating platters on a spindle. Each platter would have a read/write head on both sides, adding up to a total of 4 heads inside the assembly.

The mechanism is capable of moving the heads between the center and the outer edge of the platters in steps. The number of discrete steps that the mechanism is capable of, adds up to the number of cylinders. A cylinder describes the concentric circles that the heads would trace when they are set to a specific cylinder value. A single such circle is called a track, which in turn is divided into sectors. On our XT-era disk, the number of sectors per track would usually be 17. Coupled with the fact that a typical sector stores 512 bytes of data, we can figure out the capacity of our drive as the multiplication of the number of cylinders, heads, sectors per track and the sector size. Any triplet of Cylinder, Head, Sector also uniquely identifies a single sector on the drive assembly.

In this day and age of multi-terabyte hard disks, the CHS geometry is no longer used in practice. Nowadays our computers treat any hard drive like a big pile of sectors that they simply address from 0 to however many there are. The ATA-6 standard uses a 48-bit value for the sector number, so we'll be good until single drives reach the 144 PetaByte limit. You want to read data from a sector? Just plug its number into the drive, issue a read command and the drive does the rest. In reality the drive is still a stack of platters, heads and sectors. We just don't address them in this manner anymore.

DOSContainer aims to take the hassle out of the ancient dark art of perpar-

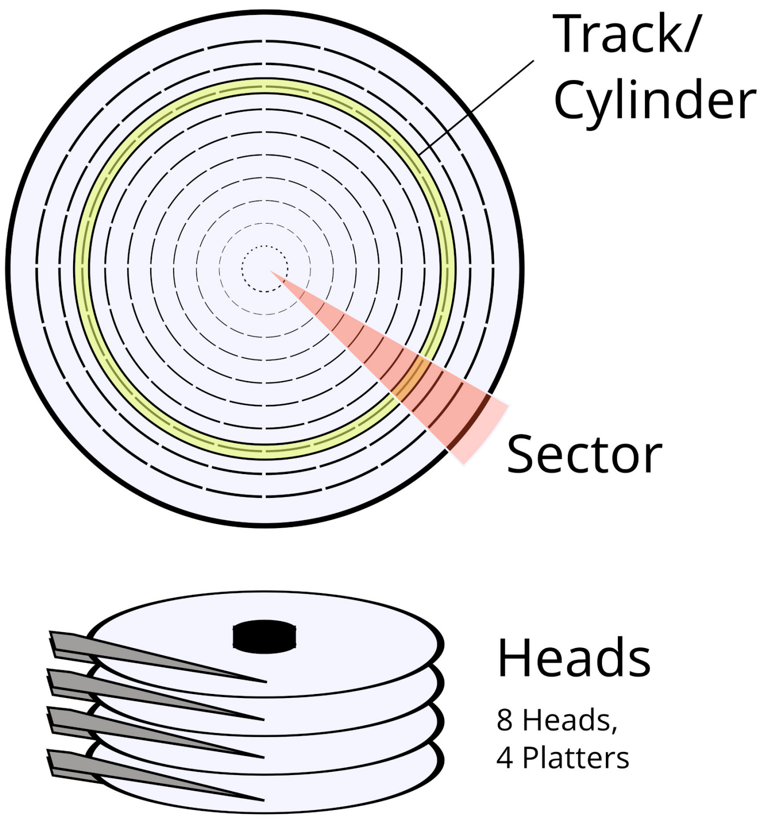

Figure 6.1: Graphical representation of the CHS geometry. Image courtesy of Lion Kimbro, copied from Wikipedia.

ing a hard disk for use with a 1980's era PC. The main purpose for this is to be able to generate bootable hard disk images for use with emulators such as PCEm, 86Box or the MiSTerFPGA AO486 and PCXT cores. These all aim to reproduce a period-appropriate facsimile of an IBM-compatible PC complete with their peripherals and clunky hard disk support of the day. It really doesn't help much that you can just plug in a VHD image and have a virtual hard drive to play with if you don't know where to even begin.

DOSContainer not only delivers a fully prepped hard disk image that is indistinguishable from a byte-for-byte copy from an actual old PC. It also gives you the needed numbers to punch into the emulated PC's BIOS so that things will actually work.

6.1 VHD versus RAW disk images

The problem for emulators is that they do not have anything physical to fall back on. A disk image file is just that: a file, a flat sequence of bytes. Such a file has no concept of "geometry" at all, so the emulation must make an educated guess about the virtual geometry it needs to present to the guest operating system. When you're running MS-DOS from 1983, that system won't know that it's talking to just a bunch of bytes that live on an SD-card over 4 decades later. So it's the emulator's job to translate the bunch of bytes back into a geometry that the host operating system understands.

In the case of really ancient IBM systems this means you only have very limited choices. The XT came with a specific 10MB hard disk that you can simulate by presenting 306 cylinders, 4 heads and 17 sectors per track. Each sector would be 512 bytes in size. Multiply all of that and you get 10653696, a figure that's pretty close to 10MB. It's just about the only drive an unmodified XT will work with. So what's to stop you from creating a raw disk image file of exactly this length and having an emulator work with it? The fact that not all emulators are created equal, that's what.

Many emulators wrongly assume a much more modern 63 sectors per track. You'll quickly notice, once you grab your calculator, that this assumption will turn up a broken number of sectors no matter how you slice or dice the number of heads and cylinders to reach the original number of bytes.

Unfortunately, with a RAW disk image, the size of the file is all you have to work with. It just stores the bytes that would go onto the original physical disk and no more. Now you *could* try and glean things from the operating system's BIOS Parameter Block, but that would mean that you actually *have* an operating system installed properly, and that you know about BIOS Parameter Block structures at all. Wouldn't it be easier if you could store things like geometry outside of the context of the operating system? That's where alternative disk image formats like VHD come in.

VHD in its simplest form is just a RAW file with a 512-byte footer tacked onto the end. The elegance of that is brilliant actually, because you'll never absorb it if you happen to incorrectly treat the VHD as RAW. Because it's only a single sector in size, it won't give the OS an additional cylinder to work with. The operating system will simply ignore it, and the footer's information is safe.

In the case of VHD we have a few fields to work with, one of which is a CHS geometry. Emulators like PCem actually pick up on this and present any geometry present in the VHD to the guest OS. This prevents the guest OS from making wrong assumptions that could end up making the disk unreadable, or worse. MiSTer treats VHD as RAW as far as I've been able to tell. That, however, does not stop DOSContainer from writing a proper VHD footer to every file it generates. It also allows DOSContainer to recognize its own image files and handle things like creation time and date.

```
Disk metadata according to VHD footer
-------------------------------------

Features           : No features
VHD Format version : 65536
Data offset        : 0
Timestamp          : 2024-04-16T16:49:30+00:00
Creator app        : DOSContainer
Creator version    : 1
Creator OS         : Windows
Original size      : 10445312
Current size       : 10445312
Geometry           : Cylinder: 150 Head: 8 Sector: 17
Disk type          : Fixed hard disk
Checksum           : 4294961133
Uuid               : 914d7f94-3826-4d1a-a1e6-d5faf5eaadb5
Saved state        : false
```

The fragment above is an example of a VHD footer's data structure as created by DOSContainer. You need not concern yourself with these details. They are all automatically managed by DOSContainer at image creation time. The fact that there is dynamic information in there like a checksum and a creation timestamp, makes each individual disk image that DOSContainer outputs slightly different.

Chapter 7

File systems

DOSContainer is being developed in chronological order with the evolution of the PC platform itself, starting with the IBM XT from 1983 which was paired with IBM PC-DOS 2.00. This operating system only supported FAT12, which is currently the only file system DOSContainer supports. As development moves forward in time, we'll encounter FAT16, FAT16B (BIGDOS), VFAT and FAT32. Those will get added to DOSContainer as appropriate.

7.1 The FAT file system

FAT was conceived in the late 1970's by Marc B. McDonald, who was Microsoft's first salaried employee. FAT would be used in Standalone Disk BASIC-80 in 1977. MS-DOS did not even exist yet at that point in time.

The FAT file system comes in a number of evolutionary steps, starting at the original 8-bit and going all the way up to 64 bits for exFAT (which is out of scope for DOSContainer). These versions are colloquially known as FAT8, FAT12, FAT16(B), FAT32 etc. But what's with those bits?

Every FAT file system is centered around a File Allocation Table. These tables behave very much like the cabinets used by post offices to sort mail.

A File Allocation Table is the table of contents for the rest of what is called a *volume*, which generally overlaps with a *partition* on a hard disk. The volume gets subdivided into *allocation units* also called *clusters*. These *clusters* are all equally-sized groups of *sectors*. A *sector* is the smallest addressable unit of allocation on a physical hard disk. Historically sectors came in different sizes for a time, but the world nowadays agrees on 512 bytes per sector. This is also what DOSContainer uses as a hard-coded value in many places across the codebase.

Now the number of bits per allocation unit determines the maximum number of these units. A *bit* is a binary digit. Without going too deep into computer science, a bit can have a value of either `0` or `1`. So if you have eight of them, you get 2^8 or 256 possible permutations. A FAT8 file system, therefore, can handle at most 256 individual allocation units or clusters on a disk. Back in 1977 that would give you 128KB of storage to work with, given that an allocation unit

Figure 7.1: A mail sorting cabinet. This is a decent enough analogy for a File Allocation Table.

would be 1 sector in size. A generous amount, given that this was a lot more than the average small computer's RAM capacity at the time.

Now you could address bigger volumes by increasing the size of the allocation unit. If you go for two sectors per allocation unit, you'd be able to store 256KB into a FAT8 file system. The problem with that, however, is in what is called *slack space*. That's the amount of empty space that is left unused when an allocation unit is larger than the size of the data that is written to it.

You could use FAT8 and design it with 32-sector allocation units. That would give you 16KB per allocation unit. But if you only ever stored 1KB files onto your drive, you would waste 15KB for each file you wrote. The larger you make your allocation units, the larger the problem of slack space becomes. This is why the number of bits per allocation unit got increased gradually over time. The first such step up was to FAT12, giving a maximum of 4096 allocation units.

7.2 Common structures to all FAT versions

All members of the FAT family of file systems have a number of traits in common. This section describes those traits.

Volume Boot Record

Starting an operating system on an old PC entailes a number of steps in sequence. When a PC is powered on, the embedded firmware or BIOS is given control. This is software that lives in ROM chips and performs actions that set up the system so that an operating system can be loaded from disk. It counts and tests things like memory, figures out where disks live and what size they are, and which one to use to load the operating system from. After figuring out where to load an operating system from, the BIOS points at the Master Boot Record on the first bootable hard disk it knows about. This is a fixed piece of data, 512 bytes in length, that contains a very small machine language program as well as a table of on-disk partitions. The machine language program is just smart enough to figure out which partition to use for startup, and where to find the operating system on that. Once it does that, it hand control over to a similar machine language routine that lives at the start of the active partition: the *Volume Boot Record*. The *Volume Boot Record* in turn knows just enough about the file system to figure out how to load the first real part of the operating system. In the case of IBM PC-DOS[1] that would be the `IBMBIO.COM` file, which must always be the very first real file written to a bootable partition, followed by `IBMDOS.COM` and `COMMAND.COM`. This triplet of files gets handed control of the CPU, with the final result being a command prompt with a blinking cursor waiting for user input provided by `COMMAND.COM`.

[1]MS-DOS as released by Microsoft itself and other OEM's would use `IO.SYS` and `MSDOS.SYS` instead of `IBMBIO.COM` and `IBMDOS.COM`. Functionally these files perform the same tasks.